Automatic Application of Power Analysis Countermeasures

Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne

Abstract—We introduce a compiler that automatically inserts software countermeasures to protect cryptographic algorithms against power-based side-channel attacks. The compiler first estimates which instruction instances leak the most information through side-channels. This information is obtained either by dynamic analysis, evaluating an information theoretic metric over the power traces acquired during the execution of the input program, or by static analysis. As information leakage implies a loss of security, the compiler then identifies (groups of) instruction instances to protect with a software countermeasure such as random precharging or Boolean masking. As software protection incurs significant overhead in terms of cryptosystem runtime and memory usage, the compiler protects the minimum number of instruction instances to achieve a desired level of security. The compiler is evaluated on two block ciphers, AES and Clefia; our experiments demonstrate that the compiler can automatically identify and protect the most important instruction instances. To date, these software countermeasures have been inserted manually by security experts, who are not necessarily the main cryptosystem developers. Our compiler offers significant productivity gains for cryptosystem developers who wish to protect their implementations from side-channel attacks.

Index Terms—Side-channel attacks, power analysis attacks, software countermeasures, compiler.

1 INTRODUCTION

Security is a fundamentally important issue in many of today’s computing platforms and applications. On average, a typical consumer used 230 embedded computing devices per day in 2008 [34], more often than not, without actually realizing it; this number is expected to exceed 1000 devices per day in the near-future [43]. Many of these devices store private data that the user does not wish to divulge, and could be exploited for malicious purposes by attackers.

Hardware and software systems must be designed with security as a high priority in order to prevent attackers from accessing confidential information. At present, hardware and software design automation tools do not treat security as a first-class design objective, despite its critical importance. In particular, side-channel attacks are a fundamental area of concern. Side-channel attacks target the physical implementation of a cryptosystem, rather than the underlying mathematical structure of the cryptosystem itself. Examples of publicly-known side-channels include power consumption [20], electromagnetic radiation [13], sound emission [32] and timing [19], among others. To perform these attacks, an adversary gains access to the device, and encrypts a statistically significant number of plaintexts, without knowledge of the value of the secret key stored within the device. The attacker measures the side-channel information as the program executes, and uses statistical methods to associate it with a subset of the bits of the key. Provided sufficient time and plaintext, an attacker can uncover the value of each and every bit of the key using this approach.

Many countermeasures have been proposed to protect against side-channel attacks; typically, a countermeasure is proposed for a specific attack (e.g., power analysis), and it seems unlikely that a universal countermeasure will be found for silicon devices. To date, these countermeasures are inserted manually by experts in side-channel protection; often, these experts are not those who implemented the cryptosystem that is being protected.

To address these concerns, we introduce a compiler that automatically applies known software countermeasures to cryptographic software to protect against side-channel attacks. The user selects which countermeasure to apply and the compiler inserts it automatically. The application of these countermeasures can significantly impact performance and code size; as a consequence, it is impractical to protect every instruction instance (shortly instruction hereafter) in a cryptographic implementation. Therefore, the objective of the compiler is to automatically identify a subset of the implementation for protection, while meeting a user-specified level of
The compilation flow for side-channel protection is a fully-automated three-step process, as shown in Fig. 1:

1) **Information Leakage Analysis** identifies the instructions that are sensitive to side-channel attacks. The compiler can analyze the traces of real side-channel measurements (power traces, in our case) if the user provides traces; alternatively, it can perform static analysis.

2) **Transformation Target Identification** looks at data dependencies involving the sensitive instructions to determine larger groups of instructions that require protection to ensure security.

3) **Code Transformation** applies the protection mechanism on instructions identified by the preceding step.

We evaluate the compiler on two algorithms, AES and Clefia, using two software countermeasures: random precharging and Boolean masking for protection. Random precharging is selected as an example of countermeasures which need local analysis and code transformations (case (I) in Fig. 1); whereas Boolean masking is a more advanced countermeasure requiring global treatment (case (II) in Fig. 1). Our results demonstrate that the compiler can automatically detect and protect all instructions that leak critical information; the resulting program ensures the same level of protection with modest run time and code-size overheads, compared to manually protected counterparts.

Although the overall flow is based on the previous work of Bayrak et al. [4], the application of global countermeasures (case (II) in Fig. 1), which was left as an open problem, is solved in this paper; Boolean masking is selected as a case study. The experimental analysis is enriched with the new countermeasure and additional benchmark ciphers to protect.

The rest of the paper is organized as follows. Section 2 discusses the related work. Sections 3, 4, and 5 respectively describe the three key steps of our compiler: Information Leakage Analysis, Transformation Target Identification, and Code Transformation. Section 6 describes our experimental procedure and presents our results applying random precharging and Boolean masking to the AES and Clefia block ciphers. The paper is concluded in Section 7.
usually publicly known. The compiler developed here focuses on power-based side-channel attacks; however, similar approach could easily be applied to other attacks, particularly EM- and acoustic-based ones, as well.

2.1 Power Analysis Attacks and Countermeasures

Power analysis attacks [20] rely on the fact that the instantaneous power consumption of most modern CMOS devices is strongly correlated with the data that is being processed at the same instant. The attacker provides different inputs (plaintexts) to the cryptosystem running on the device, and measures the real-time power consumption of each input. After the measurements are completed, the attacker statistically analyzes the relationship between the inputs and collected power traces, which depend on both the plaintext and secret key. Different methods are proposed in the literature to analyze this relationship [7], [8], [14], [20], e.g., CPA (correlation-based power analysis attack) [7] uses Pearson’s correlation coefficient. The statistical analysis allows the attacker to obtain some or all bits of the secret key.

Short after the power analysis attacks shown to be efficient and effective, many countermeasures have been introduced to protect against them. Software countermeasures include random insertion of dummy instructions, shuffling [18], [22], [38], random precharging [37], and Boolean and arithmetic masking [1], [6], [12], [28].

Historically, it has been the responsibility of hardware designers or software engineers to determine the weaknesses of a given cryptosystem and to figure out how to best apply a given countermeasure. These tasks are challenging, and requires a detailed understanding of the cryptographic algorithm at the core of the cryptosystem, the hardware platform on which the system runs, and the countermeasure mechanism. The primary objective of this work is to automate this process, so that a compiler, rather than an expert user, can apply these countermeasures; it is important to note that we do not propose any new countermeasures in this paper, nor do we attempt to compare and quantify the quality of one countermeasure versus another. The compiler introduced here will increase user productivity, and reduce potential human error that occurs during the introduction of countermeasures.

A handful of projects have looked at power analysis attacks from perspectives other than hardware design, including the Computer Aided Cryptography Engineering (CACE) [11] project. Barbosa et al. [3] have analyzed the effects of a compiler on elliptic curve cryptography. Bayrak et al. [4] proposed a framework to automate the application of power analysis countermeasures, which forms the basis of this work. Cleemput et al. [10] proposed compiler techniques to defend against timing attacks on x86 processors.

During the review process of this paper, Moss et al. [26] proposed a method to automatically apply Boolean masking. One of the main problems left open in the paper is that their method does not have control over the order of instructions in the output, leading to potentially vulnerable codes in most of today’s devices; because, the effect of masking is invalidated if two consecutive instructions use the same mask. This problem does not exist in our compiler; because, our compiler performs the transformations on assembly language programs and carefully avoids vulnerable orderings. Secondly, Moss et al. impose formatting restrictions on the high-level source code that is input to their compiler. In contrast, our compiler takes assembly language programs as input, thereby making it easier to integrate into existing compilers.

3 INFORMATION Leakage Analysis

Information Leakage Analysis is the first step of our automatic protection framework. It determines which instructions leak information that is useful for a specific side-channel attack. The input to this step is an assembly language implementation of a cryptographic algorithm that requires protection, and the output is a set of annotations that indicate the sensitivity of each instruction.

The reason that we choose to operate on assembly instructions, rather than a higher level representation, is due to the fact that the countermeasures that we apply do not alter the output behavior of the program; a strong optimizing compiler, which is unaware of this behavior, could recognize these countermeasures as redundant code, and eliminate them in an effort to improve performance and reduce code size. For example, if we apply the countermeasure at a higher-level, extra instructions added as an outcome of random precharging countermeasure to randomize the values in the data path could be removed in later stages because they have no impact on the output of the program.

Most cryptographic algorithms exhibit deterministic control flow; a common case is loops with constant iteration counts. For example, the Advanced Encryption Standard (AES) performs a fixed number of rounds (e.g., 10 for AES-128), each of which performs the same deterministic set of steps. In this work, we assumed a deterministic control and data flow in the input software.

Information leakage analysis can be performed statically or dynamically. Static analysis decouples the
assembly code into a traditional compiler intermediate representation and uses program slicing techniques [39] to identify instructions that operate on critical data (e.g., the key), and their dependencies. Static analysis requires no manual intervention from the user, however, it does not allow for platform-specific considerations, and, as a result, could be overly protective. Dynamic analysis, in contrast, exploits power traces provided by the user. Dynamic analysis exploits highly accurate device-specific leakage information, but requires the user to acquire power traces off-line. Either of these methods can provide an estimate of the sensitivity of each instruction in the program, and the user selects which option to use.

3.1 Static Analysis

Static analysis automatically analyzes a given cryptographic software implementation (in assembly language) and reports the sensitivity of each instruction; it can be enabled by --static-analysis command flag.

In the first step, the compiler automatically decompiles the input into a Control Flow Graph (CFG), with a Data Flow Graph (DFG) to represent dependencies. The compilation method is based on the Ph.D. thesis of Cifuentes [9], which is shown in Fig. 2. Unlike Cifuentes, the decompilation method produces an intermediate representation comprised of CFG and DFG, rather than a high-level language representation of the application. This makes the high level code optimizations and Cifuentes’ code generation phase unnecessary. The compiler is implemented in C++ using Lex and Yacc [21].

An idiom is a group of instructions that defines a single operation. For example, an 8-bit processor requires multiple instructions to perform a 16-bit operation, which would be grouped as an idiom (see Section 4.2 of Cifuentes’ thesis [9] for details); the use of idioms simplifies the analysis steps. The compiler constructs basic blocks so that each of them contains a single idiom. The CFG preserves the control structure of the application, and the DFG facilitates propagation of relevant information, such as sensitivity and protection requirements, through the idioms.

Next, the compiler analyzes the DFG to determine which idioms are sensitive and require protection. Since we assumed a deterministic control/data flow, a simple dependence analysis suffices in lieu of more complicated static program analysis algorithms. Let \( G \) represent the DFG, and let \( v \) denote a node of \( G \). If \( v \) accesses critical data, then \( v \) is marked sensitive; moreover, any descendent of \( v \) in the DFG is also marked as sensitive. A straightforward breadth-first search through the DFG is initiated from each node that accesses critical data. The user provides the names of critical variables as a command-line parameter to the compiler: --critical="name1",...,"nameN" (e.g., --critical="Key"). If the user does not supply the names of specific critical variables, then the compiler conservatively assumes that all variables in the program are critical.

As a motivating example, consider a load operation that copies a byte of the secret key from memory to a register; this load operation directly accesses critical data and is sensitive as a result. A subsequent instruction that \( xors \) the loaded byte with partially encrypted plaintext (called the state), is marked sensitive as well. For example, in AES, all instructions that access or modify the state are annotated as being sensitive, whereas, loop iterations and most address calculations are insensitive.

Fig. 3 provides a detailed example of how the static analysis proceeds. First, the compiler identifies the idioms and constructs the CFG and DFG; afterwards, the DFG is analyzed to determine the sensitive operations.

3.2 Dynamic Analysis

Dynamic analysis uses empirical measurements to determine the sensitivity of each instruction. The user provides an assembly language implementation of the cryptographic algorithm and power traces acquired by executing different (plaintext,key) pairs on the target platform. The user specifies a command line parameter --dynamic-analysis tracesFile to instruct the compiler to perform dynamic analysis using the power traces in the file named tracesFile. The compiler produces a set of annotations that indicate the sensitivity of each instruction.

Before running the compiler, power traces can be obtained via measurement with an oscilloscope, as explained in Section 6.1, or simulation using electronic design automation tools. The measurements are suggested to be taken at a high frequency (e.g., 4GHz in our setup), and then compressed to obtain a single power measurement for each clock cycle. High frequency sampling is to get better accuracy, whereas compression is to obtain a single sensitivity value for each instruction and for efficiency of the analysis. We used maximum extraction [23] to compress the power traces in our experiments, however, other methods such as integration or principal component analysis could also be used.
The compiler automatically performs the following steps for dynamic analysis:

i) The power traces are analyzed, and the sensitivity of each clock cycle is determined using an information theoretic metric, which estimates the amount of information that the system leaks. Clock cycles whose sensitivity exceeds a user-provided threshold are marked as sensitive. The user provides the threshold value to the compiler using the command line parameter --threshold=val, where val is a value between 0 (full protection) and 1 (no protection).

ii) The compiler associates each clock cycle from the traces with an assembly instruction. The most sensitive clock cycle associated with each instruction defines the sensitivity of that instruction.

The compiler can be used to protect the minimum number of operations that can provide security equal to a fully-protected implementation of the algorithm; thus, security is not sacrificed, but performance is improved and code size is reduced. The compiler is run once to obtain a fully protected implementation of the algorithm, and its sensitivity is determined; sensitivity of the fully-protected implementation determines the minimum sensitivity (maximum protection) achievable from the given implementation using the given countermeasure. It is important to note that the sensitivity of the entire system is dictated by its weakest element, namely the most sensitive clock cycle. The compiler is then run a second time with the sensitivity of the fully-protected implementation provided as a threshold; the compiler determines a subset of operations to protect, thereby reducing performance overhead and code size, yet not effecting the protection.

Fig. 4 provides an example: a sensitivity value is obtained for each cycle, and instructions whose sensitivity exceeds the threshold (0.4 in this example) are marked as sensitive.

**Metric for sensitivity evaluation:**

Our metric for sensitivity evaluation is based on an information theoretic metric originally proposed by Standaert et al. [35], which evaluates the resistance of a cryptographic implementation against the strongest possible power analysis attack. The metric establishes a relationship—i.e., mutual information—between the secret key that is used for encryption and the power traces. We limit the number of dimensions considered by the metric to 1, which makes it possible to simplify the formula. Since we are interested in observing the effects of single instructions, a 1-dimensional application of the metric is appropriate; higher dimensionality would be
Fig. 4. A sensitivity value, which estimates information leakage, is determined for each clock cycle, and is then associated with the assembly instruction corresponding to that clock cycle. Countermeasures will be applied to sensitive instructions (or their idioms) in the subsequent stages of compilation.

required in order to analyze higher-order effects.

Let $K$, $X$, and $L$ respectively be random variables representing the secret key, plaintext, and information leakage from the physical device which is obtained via power trace analysis; and $k$, $x$ and $l$ be realizations of $K$, $X$, and $L$ from an execution of the algorithm. Leakage $L$ is normally distributed with mean $\mu$ and standard deviation $\sigma$—i.e., $N(\mu_{k,x}, \sigma^2)$. The probability density function of $L$ is

$$N_l(\mu_{k,x}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(l-\mu_{k,x})^2}{2\sigma^2}},$$  

where $\mu_{k,x}$ represents the noiseless leakage value when $(k, x)$ pair is executed and $\sigma$ represents the constant noise standard deviation caused by the measurement. The conditional entropy of $K$ given $L$ is

$$H[K|L] = -\sum_k p(k) \cdot \sum_x p(x) \cdot \int p(l|k, x) \cdot \log_2 p(k|l, x) dl,$$

which can be rewritten as

$$H[K|L] = -\sum_k \left\{ p(k) \cdot \sum_x \left\{ p(x) \cdot \right. \
\left. \int_{-\infty}^{\infty} \left. \frac{N_l(\mu_{k,x}, \sigma^2)}{\sum_{k'} N_l(\mu_{k',x}, \sigma^2)} \right. \right\} \right\}.  \quad (3)$$

The mutual information, which quantifies the sensitivity, is $I[K; L] = H[K] - H[K|L]$. Normalizing the mutual information, $(H[K] - H[K|L])/H[K]$, makes the value independent from the number of $(X, K)$ pairs used. In the rest of the text, when we use the term mutual information, we mean normalized mutual information.

If the length of the plaintext and key are short, then it is possible to exhaustively enumerate all possible $(X, K)$ pairs to compute the metric exactly; however, this is not generally the case: for example, AES-128 has 128-bit keys and plaintexts, which would require $2^{256}$ executions. To reduce the number of traces, we can exploit some properties of large deviation theory [42]: the result obtained from a randomly chosen subset of keys and plaintexts will be close to the result obtained from exhaustive enumeration with high probability, as long as the cardinality of the subset is sufficiently large. Our experiments demonstrate that the result converges for AES-128 when we consider 16 plaintexts in conjunction with 16 keys. We tried with different numbers of pairs and we observed that the instructions that can be classified as sensitive do not change after $8 \times 8$ pairs; so we used $16 \times 16$ pairs to ensure the fidelity of the results.

4 Transformation Target Identification

Once sensitive instructions have been identified, the compiler automatically selects which instructions to protect. For relatively simple countermeasures, such as random precharging and random delay insertion, a peephole optimization suffices, meaning that each sensitive instruction can be protected atomically, independent from other instructions in the program. In this case, transformation targets are the instructions that are determined as sensitive in the previous step.

Other countermeasures, such as masking and instruction shuffling protect idioms that depend on critical data; the transformations that are applied to each idiom depend on its data and control dependencies. For example, when applying the masking countermeasure, masks are propagated between dependent sensitive idioms (i.e., the output mask of an idiom is used as an input mask of another idiom). The compiler uses a simplistic program slicing [39] technique to group the sensitive idioms that have dependencies. A forward slice contains all idioms in a program that may be affected by a given set of variables at some point in the program. For example, we can construct a slice that includes all idioms that directly or indirectly use the value of a byte of the key (the idioms that directly access this byte initiate slices). As a result, transformation targets of countermeasures that use dependencies of idioms are forward slices constructed for all of the critical data in the program.

5 Code Transformation

Lastly, the compiler applies the appropriate code transformation to the transformation targets identified by the previous step; the protected assembly language program is returned as output. The user specifies the protection mechanism using a command-line parameter --method=countermeasure, where countermeasure is either randomPrecharging (Section 5.1) or masking (Section 5.2) in the current version of the compiler.
5.1 Local Code Transformations

Some countermeasures can be applied locally to each sensitive instruction using a peephole optimization. Random precharging [37] is an example: the data path is randomly charged before and after a critical instruction using randomly generated operands. This approach is effective on devices that have high dynamic power consumption proportional to the Hamming distance between two consecutive cycles’ data flowing through a wire, gate, or functional unit; most modern embedded devices exhibit this behavior, since switching activity determines dynamic power consumption. The key idea is to randomize the bits on the critical components, such as a register or data bus; this randomizes the power consumption, since the Hamming distance between a uniformly distributed random variable and a fixed value is also uniformly random.

As an example, consider an instruction that stores the value of a variable \( x_c \) to a memory location, overwriting the value of a variable \( x_p \), which was stored there initially. The dynamic power consumption of this process is proportional to \( HD(x_c, x_p) = HW(x_c \oplus x_p) \). If both \( x_c \) and \( x_p \) have deterministic behaviors, then an attacker can exploit the dependency between the power consumption and these variables. To overcome this issue, we can store the value of a uniformly distributed random variable \( r \) before storing \( x_c \); in this case, both \( x_p \oplus r \) and \( r \oplus x_c \), which determines the dynamic power consumption of two store operations (of \( r \) and \( x_c \)), will be uniformly random. The same argument can be applied to loading a sensitive data value into a register as well.

The protection offered by random precharging and the specific operations required to perform random precharging of critical operations differ for each device, depending on its power consumption characteristics. For example, random precharging would not offer protection for devices that employ precharged busses. The selection of the appropriate countermeasure is the responsibility of the user; the compiler will deterministically apply the countermeasure, regardless of whether or not it will be effective for a given target device.

Our target device was an 8-bit AVR microcontroller. We ran some initial experiments to discern an appropriate mechanism to perform random precharging. This needs to be done once for each device to find an appropriate sequence of instructions to protect each instruction type (e.g., load, add, xor, etc.). As an example, the compiler would replace a sensitive AVR load instruction \( \text{lds Rd, ADDR} \) with the following sequence of three instructions:

\[
\begin{align*}
\text{lds Rr, RND} & \quad ; \text{assume that RND holds a random value} \\
\text{mov Rr, Rr} & \quad ; \text{and Rr is unused at the moment} \\
\text{lds Rd, ADDR} & 
\end{align*}
\]

Precise implementation details for each potential countermeasure and each target device are beyond the scope of this paper; the key point here is that some empirical measurements are necessary in order to support new countermeasures in the compiler, and to retarget the compiler to new devices.

5.2 Global Code Transformations

Some countermeasures require propagating information between dependent idioms. In this case, the compiler applies global code transformations to the slices that are constructed as explained in Section 4. Boolean masking [1], [6], [12], [28] is one of the most popular and comprehensively analyzed countermeasures against power analysis attacks that requires global code transformations and is known to provide strong protection against first-order attacks. Boolean masking xors the intermediate results of a computation with some uniformly random values (masks) so that none of the intermediate values are revealed. The masks are propagated throughout the slices, and are then removed at the end. In the discussion that follows, we use the term binary addition to refer to bitwise-xor operation.

Recall that each slice identified for protection includes a set of idioms that directly access critical data (sources), followed by their ancestors that depend on the critical data; each source has no predecessors in the slice. The compiler first applies the masks at the sources; then traverses the slice forward, propagating the masks along to intermediate nodes; it then removes the masks at the sinks at the end of slice.

Fig. 5 provides an example of mask application and propagation. Let us show how the compiler protects an intermediate node, \( v_s \), of the input program slice which performs the operation \(<y = f(x_1, x_2)>\); we use \(<\cdot\>\) notation to represent a piece of program (i.e., a node of the graph). In the protected program slice, let \( v'_1 \) and \( v'_2 \) denote the already masked versions of \( v_1 \) and \( v_2 \) respectively, where \( x'_1 = x_1 \oplus m_1, x'_2 = x_2 \oplus m_2, m_1 \) and \( m_2 \) are the propagated masks. The compiler replaces the node \( v_e \) with a masked variant \( v'_e \) that implements \(<y' = f'(x'_1, x'_2, m_1, m_2, m_3)\) and propagates the mask \( m_3 \) to the ancestors, where \( y' = y \oplus m_3 \). The function \( f' \) depends on the distribution of \( f \) over the binary addition (\( \oplus \)) operation. If \( f \) is a linear function, then it is straightforward to compute \( f' \), since \( f \) is distributive over binary addition. If \( f \) is non-linear, then specific care needs to be taken to ensure correctness. The transformations applied by the compiler for both cases are discussed below. The entire idea of masking relies on the fact that \( y' \) is statistically independent from \( y \), and that \( m_3 \) is a uniformly distributed random variable.

Unmasking is performed on the output of sinks, with respect to the program slice that is being masked. The propagated mask is added to the value computed by the sink, which generates the original result.

**Masking linear operations (i.e., xor):** Binary addition (i.e., xor) is the simplest case of masking. Consider node \(<y = x_1 \oplus x_2>\) in the unprotected graph (i.e., \( f \) is binary addition in Fig. 5). The compiler starts masking at the sources of the slice; when it reaches the binary addition
Masking requires available registers to store the masks; this information is obtained through register liveness analysis. If insufficient registers are available, then some values residing in registers are spilled (stored to memory) and reloaded after the masks are no longer needed.

**Masking non-linear operations:** Cryptographic algorithms use non-linear operations to ensure Shannon’s property of confusion [33], i.e., to obscure the relationship between plaintext and ciphertext; a substitution box, (S-box), is generally used for this purpose. Non-linear operations are typically implemented as lookup tables in software, where the index into the table depends on the input (e.g., plaintext and key). For example, in Fig. 3c, the operation \( 6 (x24 = sbox(x24)) \) is a table lookup, since the index of the array sbox is input-dependent. The other memory access operations, 3, 4 and 7, are not table lookups because the addresses that they access can be determined statically. The compiler masks a load operation as a table lookup if and only if the accessed address is input dependent. The compiler decides whether an address is input dependent by automatically applying a standard static analysis pass (constant propagation): if the queried address is a constant, it is not a table lookup; otherwise it is treated as a table lookup.

Masking non-linear operations is challenging and solutions generally require replicating the tables, one for each mask value. Our compiler uses a similar approach to Herbst et al.’s [16] for masking the non-linear operations. The node representing a sensitive table lookup operation \(<y = S[x]>\) has one parent: the node corresponding to the operation that computes \(x\); after masking, \(x\) is replaced with \(x' = x \oplus m_1\). The problem here is that we cannot express the masked output \(y' = y \oplus m_3 = S[x'] \oplus m_3\) using \(S[x']\) and linear operations; thus, we use a different masked table \(S'_m\) for each possible value of \(m\). To make this approach feasible for memory-constrained embedded systems, we limit the number of masked tables, while still ensuring a good level of protection [16], [29]. Our compiler automatically detects the minimal number of masks using the mask optimization algorithm described below.

In order to generate masked output \(y' = y \oplus m_3\) from the input \(x' = x \oplus m_1\), we use the formula \(y' = S[x] \oplus m_3 = S[x' \oplus m_1] \oplus m_3 = S'[m_3][x' \oplus m_3 \oplus m_1]\). The compiler uses Transformation 2 to generate the protected output node. Masked tables are generated using the formula \(S'_m(i \oplus m) = S(i) \oplus m, 0 \leq i < t\) and \(t\) represents the size of the table \(S\). As stated by Oswald and Schramm [29], either these tables are precomputed and stored in memory for different values of masks or a code segment that generates these tables at run time is inserted at the beginning of the implementation; our compiler supports both options. The compiler recognizes the size of a table \(i\) using static analysis of array index bound. Since we assumed deterministic control and data flow in the input software, a simple application of constant propagation suffice.
The vast majority of cryptographic algorithms are built on top of linear operations or table lookups. In principle, any other operation can be implemented as a table lookup and protected as a non-linear operation. Also, in order to avoid the cost of masked tables, some operationspecific optimizations are possible. For example, we can mask a shift operation \(<y = x \ll c,>\) by replacing it with \(<y' = (x' \ll c) \oplus m_3 \oplus (m_1 < c)>.>\) The same technique could also be applied to immediate type Boolean operations, such as \(<y = x \& 5>\) and \(<y = x \& 3>.>\)

**Optimization of number of masks:** Our compiler protects all operations that propagate sensitive information; however, using a separate mask for each operation adds a significant run-time and code-size overhead. In order to limit the overhead, our compiler automatically minimizes the total number of masks, without sacrificing protection. Different masks are assigned for all inputs \((m_1 \text{ and } m_2)\) and output \((m_3)\) of a slice node, in order to avoid the removal of masks during an intermediate operation; however, two independent nodes might share a mask without any risk.

We formalize the mask minimization problem as the edge coloring of the program slice; each input/output of a node should have different color (i.e., mask). For example, in Fig. 5(b) \(m_1, m_2\) and \(m_3\) should be different. According to Vizing’s theorem [44], the edges of an undirected graph can be colored using at least \(\Delta\) or \(\Delta + 1\) colors depending on the graph, where \(\Delta\) is the maximum degree of graph. Since the number of edges per node (degree of the graph) is at most 3 (i.e., 2 for incoming edges and 1 for outgoing edge), the optimal number of colors (i.e., masks) is at most 4 for any implementation. Our compiler implements the algorithm of Misra and Gries [25] to color the edges.

**Unmasking:** The final step is to remove the mask from the sinks of each slice, which will generate the correct (unmasked) value at the end of execution. Suppose that the sink computes \(y\) in the original source code, and calculates \(y'\) with propagated mask \(m_3\) in the protected source code, i.e., \(y' = y \oplus m_3\). Then the compiler adds the operation \(<y = y' \oplus m_3>\) to the end of the program to perform unmasking.

**Output code generation:** The final step is to convert the transformed program representation into assembly code. This step is straightforward since each node is an instruction or an idiom, which is easily revertible. The output routine reserves enough space to store the masked lookup tables and the masks used during the computations. Random numbers are generated using the hardware random number generator of the microcontroller if it exists; otherwise a code segment that generates them by standard library calls is inserted by the compiler. Lastly, the intermediate representation is traversed to generate the protected program.

### 6 Experimental Results

We use random precharging as an example of a countermeasure for which peephole optimization suffices, and Boolean masking as an example of a countermeasure that requires global code transformation for proper insertion. We selected two block ciphers to use as benchmarks: AES and Clefia. For AES, we used a hand optimized assembly [5], and a naive-C implementation. For Clefia, we used only a naive-C implementation; to the best of our knowledge, no hand-optimized assembly language implementations of Clefia are presently available. Both naive-C implementations are compiled with AVR-gcc cross-compiler using three optimization levels (i.e., -O0, -O1 and -O2s). The experimental results demonstrate that the compiler can successfully automate the insertion of the protection mechanisms into otherwise unprotected code. Our target platform is an 8-bit Atmel AVR ATmega microcontroller.

#### 6.1 Experimental Setup

Fig. 6 illustrates the experimental setup that we used to measure the power consumption. It includes a PC, the microcontroller board, a digital sampling oscilloscope, and a differential probe. The board was designed internally and we calibrated all equipment, to the best of our ability, to reduce electronic noise as much as possible. Each experiment was repeated 25 times and averaged results are reported to further eliminate random effects caused by measurement.

Power is measured across a 10Ω resistor connected in series to the \(V_{cc}\) pin of the microcontroller using the differential probe connected to a digital sampling oscilloscope. The microcontroller and oscilloscope communicate to start and stop measurements. The power traces collected by the oscilloscope are sent to the PC for off-line analysis (attacks). The PC loads software onto the microcontroller board, and verifies the results computed by the microcontroller.

#### 6.2 Random Precharging Experiments

We used our compiler to automatically apply random precharging method to the given implementations. Our experiments use the following command-line parameters, which are passed to the compiler: --dynamic-analysis traces.txt --threshold=0 --method=RandomPrecharging. We generated a file containing power traces, “traces.txt”, by running the microcontroller using randomly-generated \((plaintext, key)\) pairs, and collecting real-time power
measurements. The compiler automatically determined sensitive instructions using dynamic analysis (Section 3.2). First, we set the threshold value to zero to generate a fully-protected implementation. From there, we determined the best achievable sensitivity, which was 0.4. We re-ran the compiler using a threshold value of 0.4 to generate the partially protected implementation. Our experimental results show that the partially protected implementation is smaller and faster than the fully protected implementation, while providing similar security.

6.2.1 Security evaluation

This section reports how the protection improves security. First, we show how the compiler determines sensitive instructions by dynamic analysis. Fig. 7 reports the sensitivity of each clock cycle during the first round of the unprotected AES implementation running on the microcontroller. The regular structure of the AES algorithm leads to regular patterns corresponding to the four main AES operations: AddRoundKey (ARK), SubBytes (SB), Shift Rows (SR), and MixColumns (MC). The internal state of the AES algorithm is represented as a 4 x 4 array, and each operation acts on individual bytes of the state, leading to further regularity.

Non-linear operations, such as S-boxes, are generally the target of side channel attacks, because they highlight the difference between incorrect and correct key guesses; this increases the probability of a successful attack [30]. For microcontrollers, data transfer instructions, i.e., loads and stores, are known to leak more information relative to other instructions [23]. Fig. 7 validates these past observations. The SR operation permutes the S-box outputs via load and store instructions, and thus has the highest sensitivity peaks. MC also re-loads the S-box outputs, and has high sensitivity values as well.

Figs. 8 and 9 show the results of our partially protected implementation. Fig. 8 shows that all sensitivity values for the partially-protected implementation are below the threshold value of 0.4 during every clock cycle; Fig. 9 shows the sensitivity values for the protected code snippet from Fig. 4.

To determine the noise standard deviation, \( \sigma \), used by the normalized mutual information theoretic metric (Section 3.2), we ran a small code segment that is independent of the key and plaintext. We then computed the standard deviation of the signal for each clock cycle; the maximum value among all cycles was chosen as the noise standard deviation.

To further evaluate the security of the protection mechanism, we mounted a correlation-based power analysis attack (i.e., CPA, see Section 2.1) to the naive and protected (by random precharging) implementations of AES. We used 20,000 different power traces and calculated the correlation coefficient, \( \rho \), for both implementations. We used Hamming Weight as the power model and the output of the S-Box operation as attack point. As a result of the attack, we observed a 8.79 times decrease in correlation (see Appendix) from the unprotected imple-
TABLE 1
The number of clock cycles required to execute different executions of AES and Clefia (including key scheduling) running on the AVR microcontroller. Random precharging is automatically applied on each implementation, using two different threshold values: the pareto-optimal threshold determined as explained in Section 3.2, and zero (for full protection). Both protections have same level of security.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Original</th>
<th>Random Precharging (Partially-protected)</th>
<th>Random Precharging (Fully-protected)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Overhead</td>
</tr>
<tr>
<td>AES Optimized-assembly</td>
<td>3495 [5]</td>
<td>11131</td>
<td>7636 (218%)</td>
</tr>
<tr>
<td>AES Naive-C with -O0</td>
<td>23302</td>
<td>48092</td>
<td>2290 (90%)</td>
</tr>
<tr>
<td>AES Naive-C with -O1</td>
<td>11290</td>
<td>26212</td>
<td>14922 (132%)</td>
</tr>
<tr>
<td>AES Naive-C with -Os</td>
<td>9978</td>
<td>23299</td>
<td>13291 (153%)</td>
</tr>
<tr>
<td>Clefia Naive-C with -O0</td>
<td>148149</td>
<td>22537</td>
<td>77423 (32%)</td>
</tr>
<tr>
<td>Clefia Naive-C with -O1</td>
<td>51149</td>
<td>97117</td>
<td>45968 (90%)</td>
</tr>
<tr>
<td>Clefia Naive-C with -O0s</td>
<td>36565</td>
<td>79469</td>
<td>41904 (115%)</td>
</tr>
</tbody>
</table>

TABLE 2
The code-size (in bytes) of different implementations of AES and Clefia on the AVR microcontroller for the random precharging experiments (the size of data, e.g., S-Boxes, is not included).

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Original</th>
<th>Random Precharging (Partially-protected)</th>
<th>Random Precharging (Fully-protected)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Overhead</td>
</tr>
<tr>
<td>AES Optimized-assembly</td>
<td>802 [5]</td>
<td>2098</td>
<td>1296 (162%)</td>
</tr>
<tr>
<td>AES Naive-C with -O0</td>
<td>1354</td>
<td>2578</td>
<td>1024 (76%)</td>
</tr>
<tr>
<td>AES Naive-C with -O1</td>
<td>864</td>
<td>1698</td>
<td>834 (97%)</td>
</tr>
<tr>
<td>AES Naive-C with -O0s</td>
<td>912</td>
<td>1732</td>
<td>840 (92%)</td>
</tr>
<tr>
<td>Clefia Naive-C with -O0</td>
<td>3108</td>
<td>3912</td>
<td>2808 (90%)</td>
</tr>
<tr>
<td>Clefia Naive-C with -O1</td>
<td>1870</td>
<td>3406</td>
<td>1336 (92%)</td>
</tr>
<tr>
<td>Clefia Naive-C with -O0s</td>
<td>2350</td>
<td>4222</td>
<td>1572 (80%)</td>
</tr>
</tbody>
</table>

Fig. 9. The protected code segment from Fig. 4; the sensitivity of all instructions is now below the threshold.

mentation ($\rho = 0.422$) to the protected implementation ($\rho = 0.048$). As noted by Mangard et al. [23], the number of power traces required to mount a successful attack increases by a factor of $k^2$ if the correlation coefficient decreases by a factor of $k$; thus, we estimate that $8.79^2 \approx 76$ times more traces are required to successfully attack the protected implementation. Once again, we wish to emphasize that this factor of 76 is germane to the random precharging scheme and the correlation-based power analysis attack, not the specific compiler algorithms that are used to introduce it.

6.2.2 Performance and code-size evaluation
Table 1 reports the number of clock cycles during the execution of three different versions of each implementation: the baseline (unprotected) version and the partially and fully protected implementations. Table 2 reports the code-size in bytes. The size of data, e.g., S-Boxes, is not included in the results, since they are the same for all three versions of each implementation.

Protecting an instruction entails the insertion of additional instructions, so the fully-protected version is an upper bound on the runtime and code-size overhead that could result from an overzealous application of the countermeasure used in this study. The partially-protected version of each implementation achieves the same level of security as the fully-protected version with up to 52% runtime and 48% code-size improvement. Figure 10 shows how the threshold effects the run time for one of the naïve-C AES implementations.

The relative performance overhead due to random precharging is nearly constant across all fully-protected implementations of a specific algorithm. This is mainly due to local application of countermeasure, as each instruction is replaced with one or more instructions.

6.3 Boolean Masking Experiments
Next, we used our compiler to automatically apply Boolean masking to AES and Clefia. We executed the compiler using command-line parameters --static-analysis --critical=Key,Plaintext --method=masking. The parameters inform the compiler to use static analysis (Section 3.1)
TABLE 3
Experiments in Table 1 and 2 are repeated for the automatic application of the Boolean masking countermeasure. We used precomputed masked tables in our performance results.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Performance (clock cycles)</th>
<th>Code Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original Total</td>
<td>Masking Overhead</td>
</tr>
<tr>
<td>AES Naive-C with -O0</td>
<td>25302</td>
<td>27666</td>
</tr>
<tr>
<td>AES Naive-C with -Os</td>
<td>11290</td>
<td>13582</td>
</tr>
<tr>
<td>AES Manually Masked (Herbst et al. [16]) w/ generation of masked tables</td>
<td>9978</td>
<td>12270</td>
</tr>
<tr>
<td>AES Manually Masked (Oswald and Schramm [29]) masking only table lookups</td>
<td>-</td>
<td>12800</td>
</tr>
<tr>
<td>Clefia Naive-C with -O0</td>
<td>148149</td>
<td>154991</td>
</tr>
<tr>
<td>Clefia Naive-C with -Os</td>
<td>51149</td>
<td>57555</td>
</tr>
<tr>
<td>Clefia Naive-C with -Os</td>
<td>36565</td>
<td>43035</td>
</tr>
</tbody>
</table>

Fig. 10. Increasing the threshold reduces run-time (increases performance), but reduces the protection. Protection is equal to the threshold in the interval between two perpendicular sensitivity lines; however stays stable before min sensitivity and after max sensitivity lines.

6.3.1 Security evaluation
The security of masking, both in theory and practice, has been studied in detail, and has been shown to be resistant against first-order differential power analysis attacks [17], [24], [27], [36], [45]. Our application of masking is based on common strategies that have been suggested in the literature. Once again, our contribution is the development of a compiler that can introduce this countermeasure automatically; our results are in-line with prior work.

We first applied a sensitivity analysis on the protected implementation and observed that sensitivities of all instructions are close to 0 for the masked implementation; they are not zero because of the experimental noise.

Then, we mounted a correlation-based power analysis attack (CPA) on all implementations of AES and Clefia using 5000 power traces, and calculated the correlation coefficients. Once again, we used Hamming weight as a power model and attacked the output of the S-box operation. The correlation for the correct key never exhibited a peak neither for AES nor for Clefia algorithms, regardless of the number of traces used. Plot of the correlation for the correct key and incorrect keys as a function of the number of traces for AES implementation is shown in Appendix.

6.3.2 Performance and code-size evaluation
Table 3 reports the results of our Boolean masking experiments. Our compiler took less than one second to automatically mask each of these implementations. The performance overhead incurred by masking appears to be independent of the optimization parameter of gcc; the number of sensitive instructions is a property of the algorithm, and the optimization level cannot remove sensitive operations (as they are a required part of the algorithm) or transform them into less sensitive operations. The sensitive operations process critical data, either directly or indirectly. Non-sensitive operations include loop counter iterations, and do not benefit from masking.

There are two manually masked AES implementations for the AVR, provided by Oswald and Schramm [29] (masks only the table lookups) and Herbst et al. [16]; our compiler produces implementations with comparable results (see Table 3). Note that the given results highly depend on the factors such as the used base implementation, whether the masked tables are precomputed or generated at run time and how many masked tables are generated. A detailed analysis that explains how these parameters effect the performance is given in the paper.

to estimate the sensitivity, to treat the key and plaintext as critical data, and to use Boolean masking as the protection mechanism. The compiler automatically generated protected outputs.

Fig. 10. Increasing the threshold reduces run-time (increases performance), but reduces the protection. Protection is equal to the threshold in the interval between two perpendicular sensitivity lines; however stays stable before min sensitivity and after max sensitivity lines.
of Oswald and Schramm [29] and is beyond the scope of this paper. In the given results, our compiler and Oswald and Schramm [29]’s manual implementation uses precomputed masked tables approach, whereas Herbst et al. uses run time generated tables. Our compiler also supports generating the masked tables at run time (see Section 5.2) and an additional $11 \times t$ clock cycles are needed for each mask table generation, where $t$ is the size of table to masked. In the AES implementations, we use 2 tables: one for S-Box and one for xtime. Generating a masked table for each of them for a single mask takes 2816 clock cycles, resulting in additional 5632 clock cycles. In Clefia, we use 4 tables: 2 for S-Boxes, one for multiplication over $GF(2^8)$, and one for constant values used in key scheduling algorithm. Similar approach results in additional 11264 clock cycles for Clefia implementations. We are unaware of any prior masked Clefia implementations, so we were unable to perform a direct comparison with prior work.

Code size also depends on same parameters as the performance. Each masked table takes additional $t$ bytes and the code that generates a masked table at run time takes additional 14 bytes. For example, the overhead of 2 masked tables is 512 bytes for the AES implementation; if the user selects the compiler option to generate the masked tables at run-time, additional 28 bytes will be used, resulting in $512 + 28 = 540$ bytes overhead in total. Table 3 does not include these overheads.

6.4 Discussion

The proposed compiler does not introduce new countermeasures; instead it automatically applies known countermeasures to given software implementations of cryptographic algorithms. Hence, the improvement in security is limited by the ability of the countermeasure that has been applied, not the ability of the compiler to apply the countermeasure. None of the known countermeasures guarantee perfect security; they increase the effort required to mount a successful attack.

Our results demonstrate that the compiler is capable of identifying the most important operations to protect to limit the performance and code-size overhead. The performance analysis given here is for 8-bit AVR microcontroller; the overhead is likely to vary for different platforms. It is beyond the scope of this paper to compare the quality of different countermeasures, and it is the responsibility of the user to select the appropriate countermeasure to use.

Large parts of the methodology generalize immediately to other countermeasures; other parts, which are specific to the chosen countermeasure, do not generalize. The first two steps (information leakage analysis and transformation target identification) of our methodology are generic and can be used for the application of any countermeasure. However, the last step (code transformations) is countermeasure specific and has to be extended to support different countermeasures (e.g., shuffling).

We tested our compiler on block ciphers because of their popularity in prior work on side channel protection, availability of source code, and the applicability of countermeasures to these applications. For example, the block ciphers we chose do not employ traditional fixed- or floating-point arithmetic operations for encryption/decryption; Boolean masking alone cannot protect these operations. That being said, our methodology is not specific to block ciphers; they have simply been chosen as representative examples of the types of algorithms that our compiler could protect.

7 Conclusion

In this work, we propose a compiler which automatically applies software countermeasures to protect against power analysis attacks. The compiler can be used by software engineers who do not have any background in cryptography. As an experimental study, we have shown that our compiler was able to protect different implementations of two block ciphers, AES and Clefia. The compiler automatically determines and protects the most sensitive instructions, while obtaining comparable security to fully-protected implementations, with much less overhead. In principle, our algorithms could be implemented in any compiler that exists today.

References
